Softmax Regression

3조 한장혁 임동균 하연진

원-핫 인코딩 (One-hot encoding)

- 선택지의 개수만큼의 차원을 가짐
- 인덱스에 해당하는 원소가 1, 나머지 0

강아지 = [1,0,0]
고양이 = [0,1,0]
냉장고 = [0,0,1]

- 원-핫 벡터 무작위성

$$((1,0,0)-(0,1,0))^2 = (1-0)^2 + (0-1)^2 + (0-0)^2 = 2$$

$$((1,0,0)-(0,0,1))^2 = (1-0)^2 + (0-0)^2 + (0-1)^2 = 2$$

5.2 소프트맥스 회귀 (Softmax Regression)

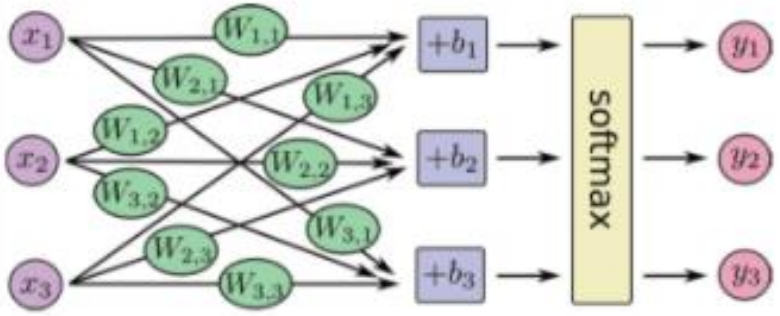
- 소프트맥스 회귀는 다중 분류를 하기 위한 기본적은 회귀

- 총 확률의 합이 1

$$p_i = \frac{e^{z_i}}{\sum_{j=1}^{k} e^{z_j}} \quad for\ i = 1, 2, \ldots k$$

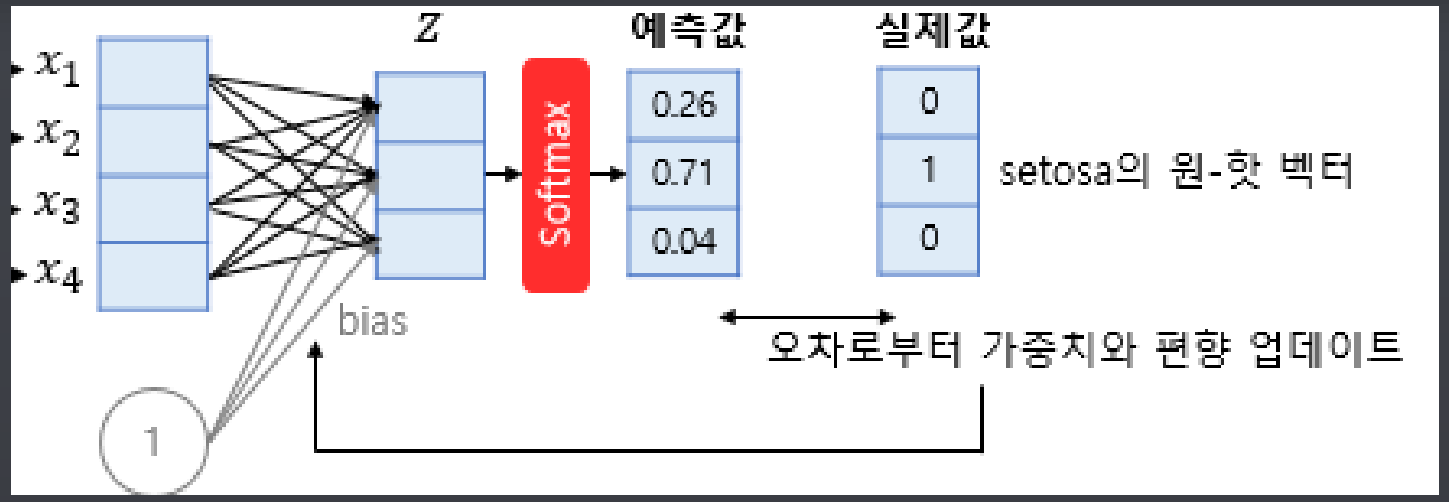
- Softmax 함수

- 소프트맥스 회귀 구조



$$\begin{bmatrix} y_1 \\ y_2 \\ y_3 \end{bmatrix} = \text{softmax}\left(\begin{bmatrix} W_{1,1} & W_{1,2} & W_{1,3} \\ W_{2,1} & W_{2,2} & W_{2,3} \\ W_{3,1} & W_{3,2} & W_{3,3} \end{bmatrix} \cdot \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} + \begin{bmatrix} b_1 \\ b_2 \\ b_3 \end{bmatrix} \right)$$

소프트맥스 회귀 (Softmax Regression)



크로스 엔트로피 함수

$$cost(W) = -\sum_{j=1}^{k} y_j \, log(p_j)$$

평균

$$cost(W) = -\frac{1}{n} \sum_{i=1}^{n} \sum_{j=1}^{k} y_j^{(i)} \, log(p_j^{(i)})$$

5.3 소프트맥스 회귀의 비용 함수 구현하기 (Low-Level)

```
import torch
import torch.nn.functional as F

torch.manual_seed(1)
```

(1) torch.rand

```
z = torch.rand(3, 5, requires_grad=True)
```

TORCH.RAND

`torch.rand(*size, *, out=None, dtype=None, layout=torch.strided, device=None, requires_grad=False)` → Tensor

Returns a tensor filled with random numbers from a uniform distribution on the interval $[0, 1)$

The shape of the tensor is defined by the variable argument `size`.

Parameters

size (*int...*) – a sequence of integers defining the shape of the output tensor. Can be a variable number of arguments or a collection like a list or tuple.

Keyword Arguments

- **generator** (`torch.Generator`, optional) – a pseudorandom number generator for sampling
- **out** (*Tensor, optional*) – the output tensor.
- **dtype** (`torch.dtype`, optional) – the desired data type of returned tensor. Default: if `None`, uses a global default (see `torch.set_default_tensor_type()`).
- **layout** (`torch.layout`, optional) – the desired layout of returned Tensor. Default: `torch.strided`.
- **device** (`torch.device`, optional) – the desired device of returned tensor. Default: if `None`, uses the current device for the default tensor type (see `torch.set_default_tensor_type()`). `device` will be the CPU for CPU tensor types and the current CUDA device for CUDA tensor types.
- **requires_grad** (*bool, optional*) – If autograd should record operations on the returned tensor. Default: `False`.

5.3 소프트맥스 회귀의 비용 함수 구현하기 (Low-Level)

(2) torch.nn.functional.softmax

```
hypothesis = F.softmax(z, dim=1)
print(hypothesis)
```

```
tensor([[0.2645, 0.1639, 0.1855, 0.2585, 0.1277],
        [0.2430, 0.1624, 0.2322, 0.1930, 0.1694],
        [0.2226, 0.1986, 0.2326, 0.1594, 0.1868]], grad_fn=<SoftmaxBackward>)
```

TORCH.NN.FUNCTIONAL.SOFTMAX

`torch.nn.functional.softmax(input, dim=None, _stacklevel=3, dtype=None)` [SOURCE]

Applies a softmax function.

Softmax is defined as:

$$\text{Softmax}(x_i) = \frac{\exp(x_i)}{\sum_j \exp(x_j)}$$

It is applied to all slices along dim, and will re-scale them so that the elements lie in the range [0, 1] and sum to 1.

See `Softmax` for more details.

Parameters

- **input** (*Tensor*) – input
- **dim** (*int*) – A dimension along which softmax will be computed.
- **dtype** (`torch.dtype`, optional) – the desired data type of returned tensor. If specified, the input tensor is casted to `dtype` before the operation is performed. This is useful for preventing data type overflows. Default: None.

소프트맥스 회귀의 비용 함수 구현하기 (Low-Level)

(3) torch.randint

```
y = torch.randint(5, (3,)).long()
print(y)
```

```
tensor([0, 2, 1])
```

TORCH.RANDINT

torch.randint(*low=0*, *high*, *size*, ***, *generator=None*, *out=None*, *dtype=None*, *layout=torch.strided*, *device=None*, *requires_grad=False*) → Tensor

Returns a tensor filled with random integers generated uniformly between `low` (inclusive) and `high` (exclusive).

The shape of the tensor is defined by the variable argument `size`.

5.3 소프트맥스 회귀의 비용 함수 구현하기 (Low-Level)

(4) One-HotEncoding (torch.tensor.scatter_)

```
# 모든 원소가 0의 값을 가진 3 × 5 텐서 생성
y_one_hot = torch.zeros_like(hypothesis)
y_one_hot.scatter_(1, y.unsqueeze(1), 1)
```

```
tensor([[1., 0., 0., 0., 0.],
        [0., 0., 1., 0., 0.],
        [0., 1., 0., 0., 0.]])
```

(unsqueeze는 특정 위치에
1인 차원을 추가하는 것)

TORCH.TENSOR.SCATTER_

`Tensor.scatter_`(*dim*, *index*, *src*, *reduce=None*) → Tensor

Writes all values from the tensor `src` into `self` at the indices specified in the `index` tensor. For each value in `src`, its output index is specified by its index in `src` for `dimension != dim` and by the corresponding value in `index` for `dimension = dim`.

For a 3-D tensor, `self` is updated as:

```
self[index[i][j][k]][j][k] = src[i][j][k]  # if dim == 0
self[i][index[i][j][k]][k] = src[i][j][k]  # if dim == 1
self[i][j][index[i][j][k]] = src[i][j][k]  # if dim == 2
```

This is the reverse operation of the manner described in `gather()`.

`self`, `index` and `src` (if it is a Tensor) should all have the same number of dimensions. It is also required that `index.size(d) <= src.size(d)` for all dimensions `d`, and that `index.size(d) <= self.size(d)` for all dimensions `d != dim`. Note that `index` and `src` do not broadcast.

Moreover, as for `gather()`, the values of `index` must be between `0` and `self.size(dim) - 1` inclusive.

Parameters

- **dim** (*int*) – the axis along which to index
- **index** (*LongTensor*) – the indices of elements to scatter, can be either empty or of the same dimensionality as `src`. When empty, the operation returns `self` unchanged.
- **src** (*Tensor or float*) – the source element(s) to scatter.
- **reduce** (*str, optional*) – reduction operation to apply, can be either `'add'` or `'multiply'`.

5.3 소프트맥스 회귀의 비용 함수 구현하기 (Low-Level)

(5) cost function

$$cost(W) = -\frac{1}{n}\sum_{i=1}^{n}\sum_{j=1}^{k} y_j^{(i)}\, log(p_j^{(i)})$$

$$cost(W) = \frac{1}{n}\sum_{i=1}^{n}\sum_{j=1}^{k} y_j^{(i)} \times (-log(p_j^{(i)}))$$

```
cost = (y_one_hot * -torch.log(hypothesis)).sum(dim=1).mean()
print(cost)
```

```
tensor(1.4689, grad_fn=<MeanBackward1>)
```

5.3 소프트맥스 회귀의 비용 함수 구현하기 (High-Level)

1. F.softmax() + torch.log() = F.log_softmax()

```
# Low level
torch.log(F.softmax(z, dim=1))
```

```
tensor([[-1.3301, -1.8084, -1.6846, -1.3530, -2.0584],
        [-1.4147, -1.8174, -1.4602, -1.6450, -1.7758],
        [-1.5025, -1.6165, -1.4586, -1.8360, -1.6776]], grad_fn=<LogBackward>)
```

```
# High level
F.log_softmax(z, dim=1)
```

```
tensor([[-1.3301, -1.8084, -1.6846, -1.3530, -2.0584],
        [-1.4147, -1.8174, -1.4602, -1.6450, -1.7758],
        [-1.5025, -1.6165, -1.4586, -1.8360, -1.6776]], grad_fn=<LogSoftmaxBackward>)
```

5.3 소프트맥스 회귀의 비용 함수 구현하기 (High-Level)

2. F.log_softmax() + F.nll_loss() = F.cross_entropy()

<Low level>

```
# 두번째 수식
(y_one_hot * - F.log_softmax(z, dim=1)).sum(dim=1).mean()
```

```
tensor(1.4689, grad_fn=<MeanBackward0>)
```

*F.nll_loss()를 사용할 때는 원-핫 벡터를 넣을 필요없이
바로 실제값을 인자로 사용합니다.

```
# 네번째 수식
F.cross_entropy(z, y)
```

```
tensor(1.4689, grad_fn=<NllLossBackward>)
```

<High level>

```
# High level
# 세번째 수식
F.nll_loss(F.log_softmax(z, dim=1), y)
```

```
tensor(1.4689, grad_fn=<NllLossBackward>)
```

5.4 소프트맥스 회귀 구현하기 (Low-Level)

2. F.log_softmax() + F.nll_loss() = F.cross_entropy()

```
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim

torch.manual_seed(1)
```

#데이터 선언

```
x_train = [[1, 2, 1, 1],
           [2, 1, 3, 2],
           [3, 1, 3, 4],
           [4, 1, 5, 5],
           [1, 7, 5, 5],
           [1, 2, 5, 6],
           [1, 6, 6, 6],
           [1, 7, 7, 7]]
y_train = [2, 2, 2, 1, 1, 1, 0, 0]
x_train = torch.FloatTensor(x_train)
y_train = torch.LongTensor(y_train)
```

#모델 초기화 & optimizer 설정

```
# 모델 초기화
W = torch.zeros((4, 3), requires_grad=True)
b = torch.zeros(1, requires_grad=True)
# optimizer 설정
optimizer = optim.SGD([W, b], lr=0.1)
```

#가설 설정 & 비용 함수 선언 및 개선

```
nb_epochs = 1000
for epoch in range(nb_epochs + 1):

    # 가설
    hypothesis = F.softmax(x_train.matmul(W) + b, dim=1)

    # 비용 함수
    cost = (y_one_hot * -torch.log(hypothesis)).sum(dim=1).mean()

    # cost로 H(x) 개선
    optimizer.zero_grad()
    cost.backward()
    optimizer.step()

    # 100번마다 로그 출력
    if epoch % 100 == 0:
        print('Epoch {:4d}/{} Cost: {:.6f}'.format(
            epoch, nb_epochs, cost.item()
        ))
```

5.4 소프트맥스 회귀 구현하기 (High-Level)

```
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim

torch.manual_seed(1)
```

#데이터 선언

```
x_train = [[1, 2, 1, 1],
           [2, 1, 3, 2],
           [3, 1, 3, 4],
           [4, 1, 5, 5],
           [1, 7, 5, 5],
           [1, 2, 5, 6],
           [1, 6, 6, 6],
           [1, 7, 7, 7]]
y_train = [2, 2, 2, 1, 1, 1, 0, 0]
x_train = torch.FloatTensor(x_train)
y_train = torch.LongTensor(y_train)
```

#모델 초기화 & optimizer 설정

```
# 모델 초기화
W = torch.zeros((4, 3), requires_grad=True)
b = torch.zeros(1, requires_grad=True)
# optimizer 설정
optimizer = optim.SGD([W, b], lr=0.1)
```

#가설 설정 & 비용 함수 선언 및 개선

```
nb_epochs = 1000
for epoch in range(nb_epochs + 1):

    # Cost 계산
    z = x_train.matmul(W) + b
    cost = F.cross_entropy(z, y_train)

    # cost로 H(x) 개선
    optimizer.zero_grad()
    cost.backward()
    optimizer.step()

    # 100번마다 로그 출력
    if epoch % 100 == 0:
        print('Epoch {:4d}/{} Cost: {:.6f}'.format(
            epoch, nb_epochs, cost.item()
        ))
```

5.4 소프트맥스 회귀 nn.Module로 구현하기

```
model = nn.Linear(4, 3)

# optimizer 설정
optimizer = optim.SGD(model.parameters(), lr=0.1)

nb_epochs = 1000
for epoch in range(nb_epochs + 1):
```

- nn.Linear 사용
- Output_dim = 클래스의 개수

- F.cross_entropy는 그 자체로 소프트 맥스 함수를 포함하므로 가설을 정의하지 않는다.
- 옵티마이저 = 경사하강법 SGD / 학습률 = 0.1
- Zero_grad = gradient를 0으로 만들기 위함

```
    # H(x) 계산
    prediction = model(x_train)

    # cost 계산
    cost = F.cross_entropy(prediction, y_train)

    # cost로 H(x) 개선
    optimizer.zero_grad()
    cost.backward()
    optimizer.step()

    # 20번마다 로그 출력
    if epoch % 100 == 0:
        print('Epoch {:4d}/{} Cost: {:.6f}'.format(
            epoch, nb_epochs, cost.item()
        ))
```

5.4 소프트맥스 회귀 클래스로 구현하기

```
class SoftmaxClassifierModel(nn.Module):
    def __init__(self):
        super().__init__()
        self.linear = nn.Linear(4, 3) # Output이 3!

    def forward(self, x):
        return self.linear(x)

'''
model = SoftmaxClassifierModel()
'''
```

```
# optimizer 설정
optimizer = optim.SGD(model.parameters(), lr=0.1)

nb_epochs = 1000
for epoch in range(nb_epochs + 1):

    # H(x) 계산
    prediction = model(x_train)

    # cost 계산
    cost = F.cross_entropy(prediction, y_train)

    # cost로 H(x) 개선
    optimizer.zero_grad()
    cost.backward()
    optimizer.step()

    # 20번마다 로그 출력
    if epoch % 100 == 0:
        print('Epoch {:4d}/{} Cost: {:.6f}'.format(
            epoch, nb_epochs, cost.item()
        ))
```

소프트맥스 회귀 클래스로 구현하기

Epoch / Batch Size / Iteration

➢ 에포크 = 트레이닝 셋이 전체 한 번 학습에 사용이 되면 한 에포크가 돌았다.

ex) 6만장이 다 트레이닝에 사용되면 한 에포크가 돌았다.

➢ 배치사이즈 = 한 번 트레이닝에 사용할 때 6만장을 어떤 크기의 묶음으로 나눌 것인가.

ex) 한 배치의 크기 = 100 / 6만장을 통해 총 600개의 배치를 얻을 수 있다.

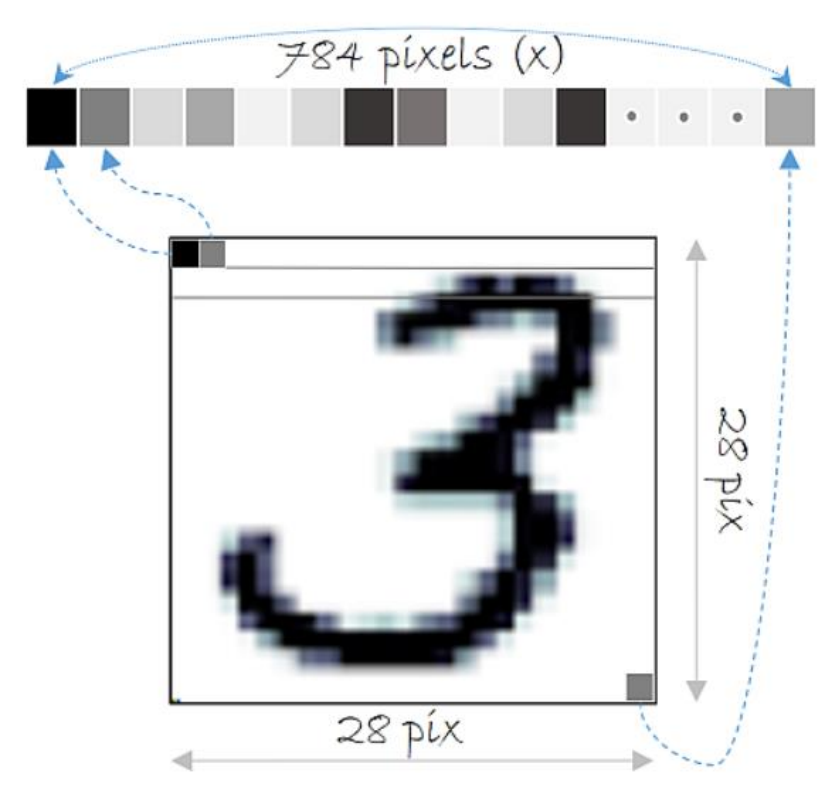
➢ 이터레이션 = 배치를 몇 번 학습에 사용을 했는가

ex) 1000개의 트레이닝 셋을 배치사이즈 500이면 2개의 배치가 있다는 것이고 2번의 이터레이션을 완료하면 1 에포크가 끝난다.

5.5 소프트맥스 회귀로 MNIST 데이터 분류하기

- MNIST 데이터란?

0~9까지의 숫자가 손글씨 이미지로 표현된 데이터셋



24 X 24 = 784차원의 벡터 생성

```
for X, Y in data_loader:
  # 입력 이미지를 [batch_size x 784]의 크기로 reshape
  # 레이블은 원-핫 인코딩
  X = X.view(-1, 28*28)
```

5.5 소프트맥스 회귀로 MNIST 데이터 분류하기

분류기 구현을 위한 사전 설정

```
import torch
import torchvision.datasets as dsets
import torchvision.transforms as transforms
from torch.utils.data import DataLoader
import torch.nn as nn
import matplotlib.pyplot as plt
import random
```

```
USE_CUDA = torch.cuda.is_available() # GPU를 사용가능하면 True, 아니라면 False를 리턴
device = torch.device("cuda" if USE_CUDA else "cpu")
# GPU 사용 가능하면 사용하고 아니면 CPU 사용
print("다음 기기로 학습합니다:", device)
```

➢ 랜덤 시드 고정

```
# for reproducibility
random.seed(777)
torch.manual_seed(777)
if device == 'cuda':
    torch.cuda.manual_seed_all(777)
```

➢ 하이퍼파라미터를 변수로 둔다.

```
# hyperparameters
training_epochs = 15
batch_size = 100
```

5.5 소프트맥스 회귀로 MNIST 데이터 분류하기

MNIST 분류기 구현하기

➢ 데이터 불러오기
torchvision.datasets.dsets.MNIST

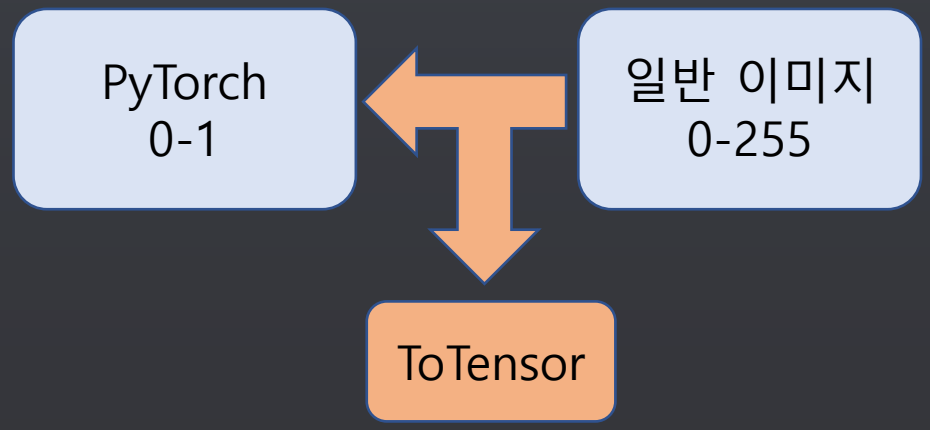
➢ 데이터 다운로드

```
# MNIST dataset
mnist_train = dsets.MNIST(root='MNIST_data/',
                          train=True,
                          transform=transforms.ToTensor(),
                          download=True)

mnist_test = dsets.MNIST(root='MNIST_data/',
                         train=False,
                         transform=transforms.ToTensor(),
                         download=True)
```

➢ 데이터로더 사용

```
# dataset loader
data_loader = DataLoader(dataset=mnist_train,
                                                batch_size=batch_size, # 배치 크기는 100
                                                shuffle=True,
                                                drop_last=True)
```



5.5 소프트맥스 회귀로 MNIST 데이터 분류하기

MNIST 분류기 구현하기

- 모델 설계
  - Input_dim = 784
  - Output_dim = 10

```
# MNIST data image of shape 28 * 28 = 784
linear = nn.Linear(784, 10, bias=True).to(device)
```

- 비용 함수, 옵티마이저 정의

소프트맥스 함수를 포함하는 크로스 엔트로피 함수

```
criterion = nn.CrossEntropyLoss().to(device) # 내부적으로 소프트맥스 함수를 포함하고 있음.
optimizer = torch.optim.SGD(linear.parameters(), lr=0.1)
```

```
for epochin range(training_epochs): # 앞서 training_epochs의 값은 15로 지정함.
    avg_cost = 0
    total_batch = len(data_loader)

for X, Yin data_loader:
        # 배치 크기가 100이므로 아래의 연산에서 X는 (100, 784)의 텐서가 된다.
        X = X.view(-1, 28 * 28).to(device)
        # 레이블은 원-핫 인코딩이 된 상태가 아니라 0 ~ 9의 정수.
        Y = Y.to(device)

        optimizer.zero_grad()
        hypothesis = linear(X)
        cost = criterion(hypothesis, Y)
        cost.backward()
        optimizer.step()

        avg_cost += cost / total_batch

    print('Epoch:', '%04d' % (epoch + 1), 'cost =', '{:.9f}'.format(avg_cost))

print('Learning finished')
```

```
Epoch: 0001 cost = 0.535468459
Epoch: 0002 cost = 0.359274209
Epoch: 0003 cost = 0.331187516
Epoch: 0004 cost = 0.316578060
Epoch: 0005 cost = 0.307158142
Epoch: 0006 cost = 0.300180763
Epoch: 0007 cost = 0.295130193
Epoch: 0008 cost = 0.290851474
Epoch: 0009 cost = 0.287417054
Epoch: 0010 cost = 0.284379572
Epoch: 0011 cost = 0.281825274
Epoch: 0012 cost = 0.279800713
Epoch: 0013 cost = 0.277808994
Epoch: 0014 cost = 0.276154339
Epoch: 0015 cost = 0.274440885
Learning finished
```

소프트맥스 회귀로 MNIST 데이터 분류하기

MNIST 분류기 구현하기_TEST

Test

```
# Test the model using test sets
With torch.no_grad():
    X_test = mnist_test.test_data.view(-1, 28 * 28).float().to(device)
    Y_test = mnist_test.test_labels.to(device)

    prediction = linear(X_test)
    correct_prediction = torch.argmax(prediction, 1) == Y_test
    accuracy = correct_prediction.float().mean()
    print("Accuracy: ", accuracy.item())
```

Thank you !